ERSS HW 4 Scalability: Exchange Matching

Koushik Annareddy Sreenath (ka266) and Ryan Mecca (rm358)

Scalability is an important factor for any modern server that will be put open to the public. There is the possibility of hundreds or even thousands of connections to be made on the server, so there are many new coding cases that need to be covered when compared to a simple program. The first of these cases is that the server does not crash in the face of many incoming requests. To handle this in our case, a well defined database (postgres) and a database manager and ORM was used (SQLAlchemy). Following the idea of not recreating the wheel, these programs have been developed in order to provide robust server software and handle thousands to millions of outstanding requests. Through using these tools, we believe that we have guaranteed basic scalability of our stock exchange server.

The next important factor to consider is concurrency. When there are many independent users and requests hitting a solid-state memory (such as a database), there is the risk of read-modify-write issues arising. These issues come when multiple users are trying to modify a row in the database at the same time, and both cannot grab the right value as they read concurrently. There are many different ways to solve this issue in a SQL database, and our group believes that we have come up with a valid and extremely scalable solution. First, we made sure that each process that the parent process spawns has its own memory stack. This was important for isolation of request handling, as well as for information accuracy. After this, the team carefully examined each database query that was made in the handling of different client requests. The queries that risked concurrency issues were marked and ultimately tagged with the “FOR UPDATE” SQl tag. This tag tells a SQL database that the row will be modified, and applies a lock to the single row. In this way, an operation can safely modify the row and then return the lock, allowing other queries to then grab the correct value and modify the row themselves. Importantly, this does not lock the entire database table, which would severely hurt the performance of the server as a whole. Only the singular row is locked, which allows for nearly all general traffic to continue. Finally, the team chose to not go with a serialization solution due to the costly rollback operation. When operations are serialized, there is a chance that two can collide and a rollback on the database will be required. This was seen as a performance hit that the team did not want to take, and therefore it was not chosen. In optimal conditions serialization might be slightly faster than row-locking, but once rollbacks are introduced our team found that this solution offered a better general solution for our use.

Next we will discuss the scalability of our server. As can be seen in Table 1, the number of CPU cores that the code ran on was varied, along with the number of concurrent requests that were made to the code. These requests were made by multiple threads all requesting to buy and sell the same stocks, so that there would be concurrency testing as well. As can be seen in Table 1 as well as Figure 1, there are clear performance increases in the code when additional cores are allocated. This is to be expected, as the OS is able to schedule processes more efficiently in this case. However, there is also a clear trend of larger performance increases as the number of orders increase. Looking at the 500, 100, and 200 order cases, it can be seen that the improvements of the server code get better and better. Differences go from approximate 1 second differences between cores to 3 second differences between core numbers. As our server was used at higher scales, we believe that this trend would continue. Higher performance will not just be gained from increasing the number of hardware cores that this server runs on, but also from intelligent coding choices made in the atomicity of the server. The data obtained as well as the graph show positive trends in both important conditions with our server code, and these trends support our conclusion that the code is well designed for scalability.

The team also ran some tests around changing the number of processes that were allocated to the server from the processor pool. We were under the assumption that more processes allocated would improve concurrent performance, which seems reasonable. However, we did not find this result when we were running our tests. We found that allocating more processes to the server had little impact on the performance time of concurrent workloads. While this result is a bit confusing, it can be explained a few different ways. First of all, compiler scheduling is a very optimized process on differing numbers of cores. We believe that the scheduling was adequate and that our tests did not affect the performance in any significant way. Additionally, it is possible that our testing did not get to a large enough scale to see this impact. We attempted to run multiple workloads in parallel, all with significant amounts of server requests, but these only numbered in the few thousands to low ten-thousands. It is possible that the server as well as Postgres and SQLAlchemy are optimized enough that significantly larger workloads are needed to allow this optimization to have an impact. While this testing did not ultimately result in beneficial results, we felt that we learned from reasoning why this could be the case (Notion says to include on this basis).

Table 1: Load Testing Results (All values averaged over 10 tests)

Number of Cores	Number of Orders				
	10	100	500	1000	2000
1	0.1824	1.6046	8.4750	17.5409	37.0060
2	0.1709	1.3698	7.2409	14.7866	33.8302
4	0.1549	1.3136	6.6470	14.0398	30.9082

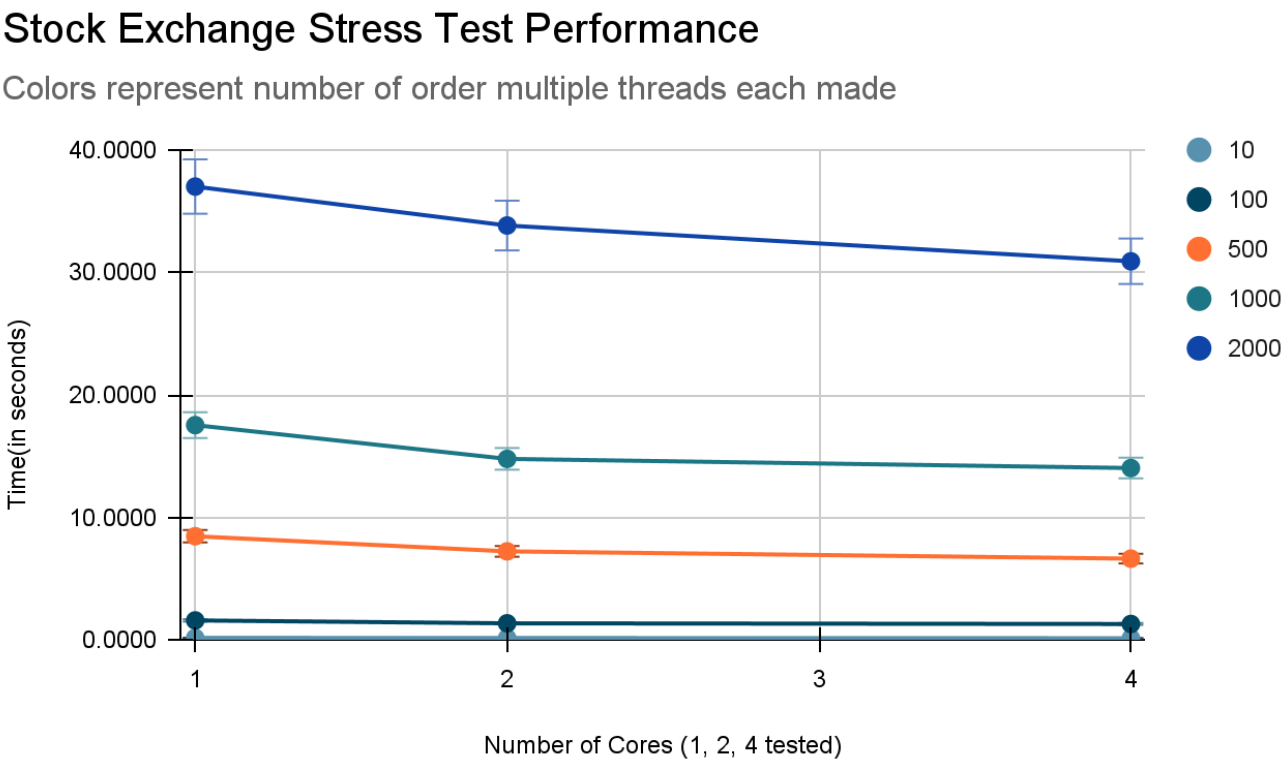

Figure 1: Graph of Load Testing Results (points averaged over 10 tests)